Scientific and Technical Report

Final Technical Report for

Design of a Parallel Object Oriented Programming Language

September 26, 1994

Sponsored by

Advanced Research Projects Agency (DOD)

Defense Small Business Innovation Research Program

Issued by U.S. Army Missile Command Under

Contract # DAAH01-94-C-R089

Name of Contractor
Scientific Computing Associates, Inc.
Business Address:
One Century Tower
265 Church Street(
New Haven, CT 06510-7010
Effective Date of Contract:
February 23, 1994
Contract Expiration Date:
September 27, 1994
Reporting Period:
February 23, 1994 - August 27, 1994

Principal Investigator:
Daya Atapattu

Phone Number:
203)777-7442
Short Title:
Parallel Object Oriented Programming Language

DISCLAIMER

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	94 Sep 26	Final 2/23/94 to 9/27/94

4. TITLE AND SUBTITLE
Design of a Parallel Object Oriented Programming Language

5. FUNDING NUMBERS
DAAH01-94-C-R089

6. AUTHOR(S)
Daya Atapattu (Principal Investigator)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Scientific Computing Associates Inc.
One Century Tower
265 Church Street
New Haven, CT 06510

8. PERFORMING ORGANIZATION REPORT NUMBER
SCADAAH01-94-C-R089-0002

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army Missile Command
AMSI-RD-PC-JB
Redstone Arsenal, AL 35898-5280

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Object-oriented programming techniques have become a vital part of modern software engineering. Most large new commercial software products are designed today using object-oriented principles along with supporting development environments and tools. Such methodology is particularly appealing for parallel computing, where there is great need for approaches capable of simplifying the programming task and producing high quality software more quickly. In this project we have carried out research that will lead to a new parallel object-oriented language based on C++.

The new language, C++-Linda®, is a novel combination of the most widely used object-oriented language and a successful environment for parallel computing that supports a virtual shared memory model for process interaction. This combination is an especially appropriate one because Linda's memory model is naturally object-oriented, enabling a seamless design that is completely consistent with object-oriented methodology. In this Phase I project, we have carried out research to understand the basic issues underlying parallel object-oriented languages, and we have created a preliminary design for C++-Linda. In a future Phase II project, we plan to develop a prototype implementation that will serve as a guide for eventual commercialization.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel computing, Programming languages, Object-oriented languages, Linda, C++-Linda, C++	20
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

Final Report

Contract Number: DAAH01-94-C-R089, ARPA SBIR Program

Design of a Parallel Object-Oriented Programming Language

Scientific Computing Associates, Inc.
One Century Tower
265 Church Street
New Haven, CT 06510-7010

Daya Atapattu, Principal Investigator
Phone: (203) 777-7442 Email: atapattu@sca.com

September 27, 1994

1 Introduction

This project undertook the task of producing a design for an object-oriented parallel language. In particular, we proposed that Linda® [1] technology be combined with an object-oriented language to achieve this objective. This report presents the results of our Phase I work.

We found that Linda's approach to the coordination of parallel processes is a good conceptual match to the object-oriented model of computation. A running object-oriented program can be regarded exactly as a collection of persistent, heterogeneous, evolving objects containing information. This is completely consistent with the view taken in Linda, where processes interact through evolving data objects (called "tuples") residing in a logically-shared associative memory called tuple space. When a process has information to communicate to other processes, it generates a tuple and drops it into tuple space. Processes that need information read or withdraw tuples to obtain it. The content of tuples is arbitrary (supporting essentially the full range of data types in any underlying base computational language such as C, FORTRAN or C++), and the means of access is quite general, permitting discrimination

[1] Linda is a registered trademark of SCIENTIFIC Computing Associates, Inc.

based on structure, or content, or both. As a result, it is natural to frame communication in a parallel object-oriented program directly in terms of tuples. In this project, our attempt has been to generalize Linda's tuples into abstractions of objects from object-oriented technology, thus providing the basis for an extremely powerful and natural parallel object-oriented programming language.

Our task for Phase I of the SBIR was to study the design of a parallel object-oriented language. Our specific objectives were to:

1. Determine the most appropriate base object-oriented computation language;
2. Design a tuple space structure supporting object-oriented programming data abstractions;
3. Examine the relationship of class abstraction (including the concept of inheritance) to the tuple space model; and
4. Identify the key issues underlying an efficient implementation of the proposed Linda parallel object-oriented programming design.

We started by studying other research projects that are involved in designing parallel object-oriented languages. We next spent a considerable amount of time in evaluating the choice of base language. Following this, we carefully examined Linda's tuple space structure in the context of the practices of object-oriented programming. Finally, we created a prototype design for a C++-Linda language which should provide a very natural and effective syntax and semantics and which should be both consistent with the principles of object orientation and implementable efficiently. In summary, we accomplished completely the objectives of the Phase I project.

In the balance of this report, we describe our work in more detail. We begin with discussions of object-oriented programming methodology and Linda's virtual shared memory technology. These focus especially on the management of software complexity using object-oriented technology and the achievement of increased performance from parallelism, and they include brief comments on several other research efforts in this area. Following this, we describe the work on each of the four stated objectives. Finally, we go through a detailed description of the prototype design for the C++-Linda language.

2 Parallel Object-Oriented Paradigm

In recent years, object-oriented programming has emerged as a way of reducing the complexity of design and development of software and of enhancing software reuse. The basic components of object orientation can be listed as encapsulation, abstraction and polymorphism. Encapsulation groups an object's state (data) and operations (functions) and provides only a well-defined interface to an object; the program does not depend on hidden

details. Abstraction simplifies program design by providing a mechanism to group similar objects together. Polymorphism provides a way for abstractions to overlap and intersect.

Parallel computing requires much of the parallel programmer. This programmer must manage the full range of complexity facing the programmer of sequential computers. In addition, issues unique to parallel computing must be addressed. These issues include:

1. Process management
2. Synchronization
3. Interprocess communication

A programming environment for parallel computing must address each of these matters. Most approaches tightly bind the management of the parallelism with the computation into a single programming environment. This forces the programmer to manage simultaneously both components of the parallel programming task, which complicates the programming effort.

We believe that a better approach is based on coordination languages. A coordination language is one that coordinates only the interaction of processes, and leaves the computational portions of a program to a familiar language from sequential computing. Since a coordination language *is* a language, there is a compiler to provide syntactic support and increase the flexibility of the programming environment. Because of their small size, coordination languages are easy to learn. In addition, they support a level of uniformity across computing languages which can significantly enhance programmer efficiency.

The most well known coordination language is Linda. Linda joined with either C or Fortran is supported commercially by Scientific Computing Associates, Inc. It has thousands of users around the world, and it has become one of the common parallel programming environments in use today.

Linda consists of a small number of powerful operations that may be integrated into a conventional base language, yielding a dialect that supports parallel programming. Thus, for example, C and FORTRAN with the addition of the Linda operations become the parallel programming languages C-Linda and FORTRAN-Linda. Scientific Computing Associates, Inc. has developed a number of parallel programming systems based on Linda. Each includes a preprocessor, to translate from a Linda parallel language (C-Linda or FORTRAN-Linda) into the corresponding base language (C or FORTRAN), possibly with the use of some automatically-generated auxiliary Linda routines, and kernel libraries to support the Linda operations at runtime. Portability comes from the consistency of the preprocessor between systems, while efficiency comes from the use of native C and FORTRAN compilers for the actual generation of object code, and hardware-specific implementations of the kernels. Commercial versions of Linda now run well on a broad range of parallel computers, from shared-memory multiprocessors, to distributed-memory machines such as hypercubes, to networks of workstations.

In this project we have studied the design of C++-Linda. In our design we do not simply implement parallelism in an object-oriented language, but introduce parallelism in an object-oriented way. We found that Linda's virtual shared memory based technology is especially

well suited for this. An important advantage of Linda is that it provides a tuple-space-based parallelism in a way that is independent of the base language. Because of this the programs written in the base language do not have to be re-designed—usually destroying the object-oriented design—in order to be combined with Linda. Moreover, Linda allows programs to access tuple space objects through only six operations, providing encapsulation and a well-defined interface. In fact, even the Linda implementations of non-object oriented languages introduce benefits of object orientation. This is because Linda's primary data structure, *tuples*, can be considered as objects (even though they do not strictly fall within the common programming-language definitions of "object") providing abstraction and encapsulation. Our proposed language illustrates how these properties of Linda are combined with an object-oriented base language to achieve true object-oriented parallelism.

As a part of understanding "object-oriented parallelism" we studied some of the research projects that are involved in parallel oriented parallel languages. pC++ [5] is a parallel C++ language largely oriented towards data-parallel programs, and is based on concurrent aggregates[4] and High Performance Fortran (HPF) ideas. The HPF style in pC++, in which explicit data distributions are used to create "distributed objects," contrasts with the task parallel nature of C++-Linda. While this may fit many parallel programming problems, in other settings, users have found Linda implementations to be far easier to use effectively than constrained HPF-based solutions.

Computational C++ (CC++)[3] is another parallel C++ language under current development. In CC++, data migration is done at a lower level (current implementations use PVM[6]), giving CC++ the same feel as message-passing-based systems for C or FORTRAN parallel programming. Process creation is done through "global pointers" that are obtained by placing an object explicitly on a different node. In both respects, the CC++ language places significantly more burden on the programmer than is desirable. Experience from parallel programming projects using C and FORTRAN suggest that the greater the effort required from the programmer, the longer and more difficult is the program development cycle.

While pC++ and CC++ were the projects studied most carefully in this work, we note that there are a large number of research efforts under way on parallel object-oriented programming. Mentat[1], CHARM++[7] and COOL[2] are other noteworthy projects.

3 Progress on Specified Objectives

As noted above, our SBIR proposal laid out four principle objectives:

1. Determine the most appropriate base object-oriented computation language;

2. Design a tuple space structure supporting object-oriented programming data abstractions;

3. Examine the relationship of class abstraction (including the concept of inheritance) to the tuple space model; and

4. Identify the key issues underlying an efficient implementation of the proposed Linda parallel object-oriented programming design.

Most of our effort went into meeting the first and last objective. It turned out that our prototype design for C++-Linda did not involve significant change in the tuple space structure from other non-object-oriented Linda systems. As a result, we spent less time than anticipated on the second and third objectives while still completing all of the specified work. The end result of this Phase I project is a language design in which a set of high-level concepts are used to introduce parallelism to C++ in an very natural and graceful fashion. At the present time, there are no commercially-available parallel object-oriented languages (though a few claim to provide some parallel capabilities in an *ad hoc* way). There are a few university research projects addressing parallel object-oriented languages, and we have studied them carefully to assess their technology. Our final prototype design, however, does not resemble any of these existing projects, but is a marriage of "object orientation" and Linda's "virtual shared memory methodology."

The remainder of this section describes the progress we made on the listed objectives.

3.1 Base Language

The basic factors that guided our selection of a base language are: the technical suitability of the language, popularity among software developers, availability of efficient implementations of compilers, and availability of a suitable front-end parser for our prototype and development work. After considering Smalltalk, Eifel and C++ we decided to use C++ as our base language.

One of the advantages of C++ is that the language has a strong typing system. This is different from most other object-oriented languages such as Smalltalk that use a weaker type system. Weak type systems combined with dynamic bindings do have some advantages: for example, the same program code can be used with variables of different types. However, we believe that strong static typing helps the development of parallel programs by detecting typing errors at compile time. This is important because typing errors in parallel programming can be hard to detect at run-time, since programs may simply deadlock. Strong typing also enables our compiler to do better analysis at compile time and generate more efficient code.

The popularity of C++ is no secret, and it is, in fact, the most widely used object-oriented programming language among all software developers today. This is evident both from an examination of the technical literature and trade press, and from the large number of research projects on parallel C++ languages now underway.

C++ has inherited efficient compilation techniques from C. There are very good commercial compilers for C++ available today. Availability of quality optimizing compilers is important because it is a requirement for high performance applications. Moreover, because C++ compilation is so closely tied to C compilation, we expect that our strong base of knowledge on parallel C should help us with development of suitable techniques for C++.

The front-end of the Linda compiler requires a modified base-language parser to recognize Linda-specific context and convert it into invocations of base-language functions. In the case of C-Linda, we use the parser from the GNU C compiler for this purpose. Since the GNU product has now evolved into a combined C/C++ compiler, we anticipate that we should be able to adapt the newer version of the GNU parser to our needs for C++-Linda. If this is possible, it has an additional advantage in that our familiarity with GNU parsers should cut down the time required to modify it.

3.2 Tuple Space Structure

In the Linda model, the tuple space provides a virtual shared memory. The user interacts with the tuple space explicitly using six tuple transfer and matching operations. In C-Linda and Fortran-Linda we have implemented a single logical tuple space for process communication and thread creation. In an object oriented environment, where data access and visibility are more finely controlled, a single "flat" tuple space may break encapsulation. We investigated the advantages and disadvantages of creating multiple tuple spaces to support the stricter scoping and encapsulation provided by C++.

Explicit multiple tuple spaces seemingly provide a clean solution where objects of different scope can be kept separated. Management of multiple tuple spaces, however, would be cumbersome, and such a system could introduce an extra layer of complexity and overhead to the underlying run-time system. Moreover, we see no advantage to making the multiplicity of tuple spaces visible to the user; they would only increase complexity without providing enhanced functionality in most cases. In any event, tuple matching procedures (such as those already used for efficiency reasons in C-Linda) can help to insure that objects of different scopes do not interact with one another. This should provide the essential functionality of (hidden) multiple tuple spaces while retaining the simplicity of a single user-visible tuple space.

3.3 Incorporation Of Class Abstraction In Tuple Space

Abstraction and inheritance are important characteristics of object-oriented programming, so the tuple space model must be consistent with these concepts. Abstraction allows the programmer to conceptualize ideas at a higher level without dealing with underlying complexities. Inheritance enhances this concept by supporting relationships between abstractions. In designing a parallel language we have to support these features to facilitate parallel program design within the object model. At first blush, it appeared that we would have to model explicitly the class hierarchies implied by abstraction and inheritance. Therefore, we considered the possibility of constructing hierarchies of tuple spaces to match the class hierarchies. Such a disjoint group of tuple spaces, we thought, might support abstraction and inheritance in a straightforward manner.

As the language design proceeded, however, we came to a better understanding of the programmer's environment. While C++-Linda programmers may design applications with

hierarchical classes, they really need not know about the detailed structure of the tuple space. Supporting the hierarchical structure of abstractions within tuple space can and should be the responsibility of the Linda system. This led us to design a C++-Linda language using a single tuple space in which compile-time and link-time analysis will be used to provide the support needed for abstraction and inheritance without burdening the programmer with additional complexity.

3.4 Implementation

The implementation issues of C++-Linda can be broadly classified into two categories: basic software component implementation, and C++ and object-oriented-specific issues.

The basic higher level structure of C++-Linda should be identical to that of C-Linda. As discussed under Section 3.1, we believe that it will be relatively simple to implement a suitable front-end parser. As in other Linda systems we will need an analyzer so that tuples are statically classified for run-time efficiency, and a C++-Linda kernel that provides run-time support.

The design of the Linda kernel is complicated by the requirement of knowing user defined types at the kernel level. In previous Linda systems, the kernel needed to know only the built-in types of the data, since C structures and Fortran common blocks were considered chunks of bytes. In C++-Linda, however, we have to interpret objects at kernel level for matching and copying purposes. This implies that part of the Linda kernel has to be compiled with the user program with user declarations of classes. Either our front-end parser must collect the relevant class declarations, or the user will have to put them in a header file; the more suitable method can only be determined after a prototype development.

We also expect some problems due to differences in implementations of C++ objects. For example, we will be able to make no assumptions about the structure of an object as each compiler is free to determine its own format. These problems can be alleviated by wrapping each object in a descriptive layer as in most distributed object management systems.

4 Language Description

We believe that there are two kinds of C++ programmers: programmers who use C++ as an extension of C, and object-oriented programmers. This project is clearly targeted towards the latter group; but the programmers of first category are probably in the majority, and C++-Linda must cater to their requirements as well. Non-object-oriented C++ programmers often depend heavily on the fact that the C++ language is (almost) a superset of ANSI C. As one of our design criteria, we have followed this language relationship and made C++-Linda a superset of C-Linda. All the C-Linda constructs are valid in C++-Linda. When a C++ language feature that is not present in C is involved, we provide new syntax and semantics.

Moreover, we provide a new set of constructs for object-oriented programmers. These "object-oriented specific" constructs are consistent with the other syntax that we inherited from C-Linda. In a strict object-oriented design, the programmer deals exclusively with

objects, so the ability of direct manipulation objects is necessary. In this model, tuple space effectively becomes an "object space," and the programmer creates new processes using member functions of objects. In this way, we extend the flexibility (ability to program in object-oriented style or non-object-oriented style) provided by C++ to C++-Linda users.

4.1 Data Communication and Synchronization

Linda augments the base language with six simple operations: `in()`, `out()`, `rd()`, `eval()`, `inp()`, and `rdp()`. The operation `eval()` does process creation and is discussed in the next section.

The basic data structure in Linda is a *tuple*, an ordered, typed set of data items. For example,

```
("tag", 20, x, arr:20)
```

is a C-Linda tuple with four data fields. The notation in the fourth field refers to the first 20 elements of array `arr`.

The `out()` operation causes a tuple to be generated and added to the tuple space. The `in()`operation searches for a matching tuple in the tuple space, blocks until it finds one, copies specified data into the variables specified by the `in()`, and removes the matching tuple from the tuple space. The `rd()` statement is similar to `in()` except that it does not remove the matching tuple. The operations `inp()` and `rdp()` are similar to `in()` and `rd()`, but they do not block if a matching tuple is not present. They return a status indicating whether a matching tuple was found.

In C++-Linda, the programmer places an object in tuple space using the construct:

```
obj.out()
```

where `obj` is the name of an object.

The member function `out()` takes no arguments and is usually defined by the user. When the `out` function is not defined by the user, the compiler generates a default function that copies all the data members to the tuple space object. However, the default function does not include any pointers that are class members. For example, the default function for the class,

```
class array {
public:
   array(int n) : size(n) { a = new int[n]; }
private:
   int *a;
   int size;
}

out(size);
```

just copies the size of the aggregate pointed to by "a", but not a[]. This is because without user definitions, the compiler cannot determine the exact shape and size of the data pointed to by a pointer. The tuple space typically resides in a different address space, and therefore copying the local address is not very useful. When the user intends to copy data attached to a class by a pointer member, it is necessary that the member function out() be defined.

The class designer uses Linda syntax in defining the member function out(). This frees the user from concentrating on the efficiency of the transfer scheme; the user specifies what data is to be transferred, and the Linda system does the transfer by the most efficient method on the particular architecture.

For example in the class:

```
class array {
public:
   array(int n) : size(n) { a = new int[n]; }
   out(a:size);
   in(?a:);
private:
   int *a;
   int size;
}
```

the member function out() describes size elements of the array pointed to by "a" as the data to be copied into the tuple space object.

Alternatively, the function out can be defined outside the class declaration as in:

```
array::out(a:size)
```

The out() statement, when used on an object, results in instantiating a complete object (a "tuple object") in the tuple space, but data members not explicitly specified in out() will have initial values in the tuple space object (current values are not copied). An out() statement with no fields directs the compiler to create an object in the tuple space, but not to copy any data fields from the invocation object to the tuple space object.

The Linda operations in() and rd() can be used to copy data from the tuple space. In the case of in() the corresponding tuple is removed from the tuple space as well.

C++-Linda uses:

```
obj.in()
```

and

```
obj.rd()
```

for these operations.

The operations in() and rd() on objects match only with tuple objects of the same class.

Again, `in()` has to be defined as an argument-less member function by the user; when not defined a default definition that copies all the data members, but not the pointer members, is generated by the compiler. For example the default `in()` function for the class `array` is:

```
in (? n)
```

The statement `obj.in()` searches for an object of the same class as `obj` that matches with data values specified in the `in()` function, copies data members specified with a "?" (linda formals), and removes the corresponding object from the tuple space.

The `obj.rd()` is similar to to `obj.in()`, but it does not remove the object from the tuple space. Definition of `rd()` is optional, when not defined the compiler defaults to the field descriptions given in `in()`.

For example, the user can define the `in` function as:

```
array::in(size, ? a:)
```

to match and copy data from a tuple object that has exactly `obj.size` elements in its "a" array.

Note that `in()` of a class can specify different members than `out()`. There is also a restriction that only class members can appear in member functions `in()`, `out()`, and `rd()`.

Note that in the above scheme, we always refer to and manipulate whole objects; parts of objects are never used. It is possible to match on a part of an object and copy another part, but the objects always stay intact. The matching and copying parts of an object is tied to the class, extending the class abstraction to include the Linda operations.

C++-Linda also supports standard C-Linda syntax for `out` and `in`. In C-Linda type syntax, the user has flexibility of working with parts of objects as long as the parts are accessible according to C++ access rules. This style allows us to to use:

```
out("tag", v, obj.member)
```

or

```
out("tag", obj.method(args))
```

When members of objects are directly addressed in this fashion, the member function `out()` is ignored, and a C-Linda style tuple is generated in the tuple space.

When an object name, not a part of an object, appears as a field of a C-Linda style `out()` or `in()`, however, Linda use the member functions `out()` and `in()` of the corresponding class to determine the data members to be copied.

Therefore,

```
out(obj)
```

has same semantics as,

```
obj.out()
```

4.2 Process creation

Creation of processes (or threads) is a basic requirement of any parallel programming system. Linda systems (C and Fortran) provide an *eval* statement for this purpose. For example, the C-Linda statement:

```
eval("tag", foo(foo_arg1, foo_arg2), bar(bar_arg1))
```

creates processes to evaluate each of the three fields and immediately returns. (In practice, `"tag"` trivially evaluates to itself and only two threads are created for `foo()` and `bar()`.)

The primary thread creation construct of C++-Linda is:

```
obj.eval()
```

where `obj` is an object. The above statement creates an object of same type as `obj` in a different node (or thread), copies data specified by `out()` statement to the newly created object and cause the evaluation of one or more member functions of new object. The member functions to evaluate are specified by the user by prepending the keyword *eval* to function declarations in the class. If there are multiple functions marked for *eval* they will execute concurrently disregarding any race conditions. The functions marked *eval* must not take any arguments, and when used to create a thread, the return value of a such function is not accessible to the program. Therefore, the programmer must arrange the return value to be stored in a data member.

The following code segment illustrates how the dot product of two vectors can be computed using `eval`.

```
class DotProd {
public:
    DotProd(int n)
        : size(n) {
              vec1 = new float[n];
              vec2 = new float[n]; }
    eval void DoDot(); // result = vec1 . vec2
    out(vec1:n, vec2:n);
    in(? result);
    void InitVec(float *data1, float *data2);
    float GetResult() { return result; }
private:
   float *vec1;
   float *vec2;
   float result;
   const int size;
};

DotProd dp1(100);
```

```
dp1.InitVec(ptr1, ptr2); // intialize vec1[] and vec2[]
dp1.eval();
...
dp1.in();
r = dp1.GetResult();
```

Note that definitions `out()`, `in()`, and the keyword *eval* are required here. The `out()` member function specifies the data items and sizes to be copied to the object in the tuple space, the *eval* keyword denotes the function to be executed in a different node, and `in()` specifies the variable to be copied back as the result of the computation. The vectors `vec1[]` and `vec2[]` have not changed and they are not copied back.

The C-Linda constructs also can be used for thread creation.

For example,

```
eval("tag", foo(arg1, arg2))
```

executes the function `foo()` in a different thread.

Similarly, one can use:

```
eval("tag", obj.bar(arg1, ...)
```

to execute a member function `bar()` of an object `obj`. Here the keyword `eval` in the declaration of `bar` is not necessary.

Also,

```
obj.eval()
```

and

```
eval(obj)
```

are semantically identical.

4.3 An Example

Here we expand our code segment for computation of dot product to a parallel matrix multiplication. In this example, the function `Matmul` takes two square matrices of size n, and places the product matrix in a matrix pointed by the third argument. Note that this is not the most efficient method to do a matrix multiplication C++-Linda. However, this is a natural extension of the dot product example, and is shown here for illustration purposes.

```
class DotProd {
public:
    DotProd(int n)
        : size(n) {
```

```
                row_vec = new float[n];
                col_vec = new float[n]; }
        eval void DoDot(); // result = vec1 . vec2
        out(row_vec:n, col_vec:n, row_num, col_number);
        in(? row_number, ? col_number, ? result);
        void InitVec(float **a, float **b, int row, int col);
        float GetRowNumber() { return row_number; }
        float GetColNumber() { return col_number; }
        float GetResult() { return result; }
    private:
        float *row_vec;
        float *col_vec;
        int   row_number;
        int   col_number;
        float result;
        const int size;
    };


    DotProd::InitVec(float **mata, float **matb, int row, int col)
        : row_number(row), col_number(col)
    {
        for (int i = 0; i < size; ++i) {
            row_vec[i] = ((float *)mata)[row*size+i];
            col_vec[i] = ((float *)matb)[i*size+col];
        }
    }


    DotProd::DoDot()
    {
        float r = 0.0;

        for (int i = 0; i < size; ++i)
            r += row_vec[i]*col_vec[i];
        result = r;
    }


    Matmul(float **mata, float **matb, float **matr, int n)
    {
       DotProd *dp = new DotProd(n);
```

```
    for (int i = 0; i < n; ++i) {
       for (int j = 0; j  < n; j++) {
          dp->InitVec(mata, matb, i, j));
          dp->eval();
       }
    }
    for (int i = 0; i < n; ++i) {
       for (int j = 0; j  < n; j++) {
          dp->in();
          matr[dp->GetRowNumber()][dp->GetColNumber()]
                 = dp->GetResult();
       }
    }

}
```

5 Compilation

The C++-Linda implementation should be similar to the C-Linda implementation, except that we will have to add additional facilities for operations on objects. Here we begin with a description of the C-Linda compiler, and then go on to specifics about how we would build a C++-Linda compiler.

5.1 C-Linda Compilation

Linda adds very little syntax to the sequential computational language to provide effective facilities to create and manage parallelism. However, Linda's simplicity, combined with its powerful associative matching semantics, means that it requires great care and sophistication to produce effective implementations. A Linda implementation involves three basic components: a language-dependent pre-compiler, a link-time optimizer, and an architecture-dependent run-time library. We'll describe now how these fit together to achieve run-time efficiency. Figure 1 illustrates the compiling and linking process for SCIENTIFIC's C-Linda product. (The one for Fortran-Linda is similar.) The pre-compiler processes C-Linda source code to produce pure C modules in which the tuple space operations are replaced by calls to functions which will, in turn, invoke routines in the run-time library. (These intermediate functions are generated automatically during optimization at link time.) The pure C modules are then compiled using the native sequential language compiler. In the course of this processing, the pre-compiler collects information about tuple space usage which is saved in a Linda object file along with the base language (C, in this case) object code.

At link-time, the Linda pre-linker analyzes tuple-space accesses over all Linda operations used in a complete program and fills in the bodies of the intermediate functions mentioned

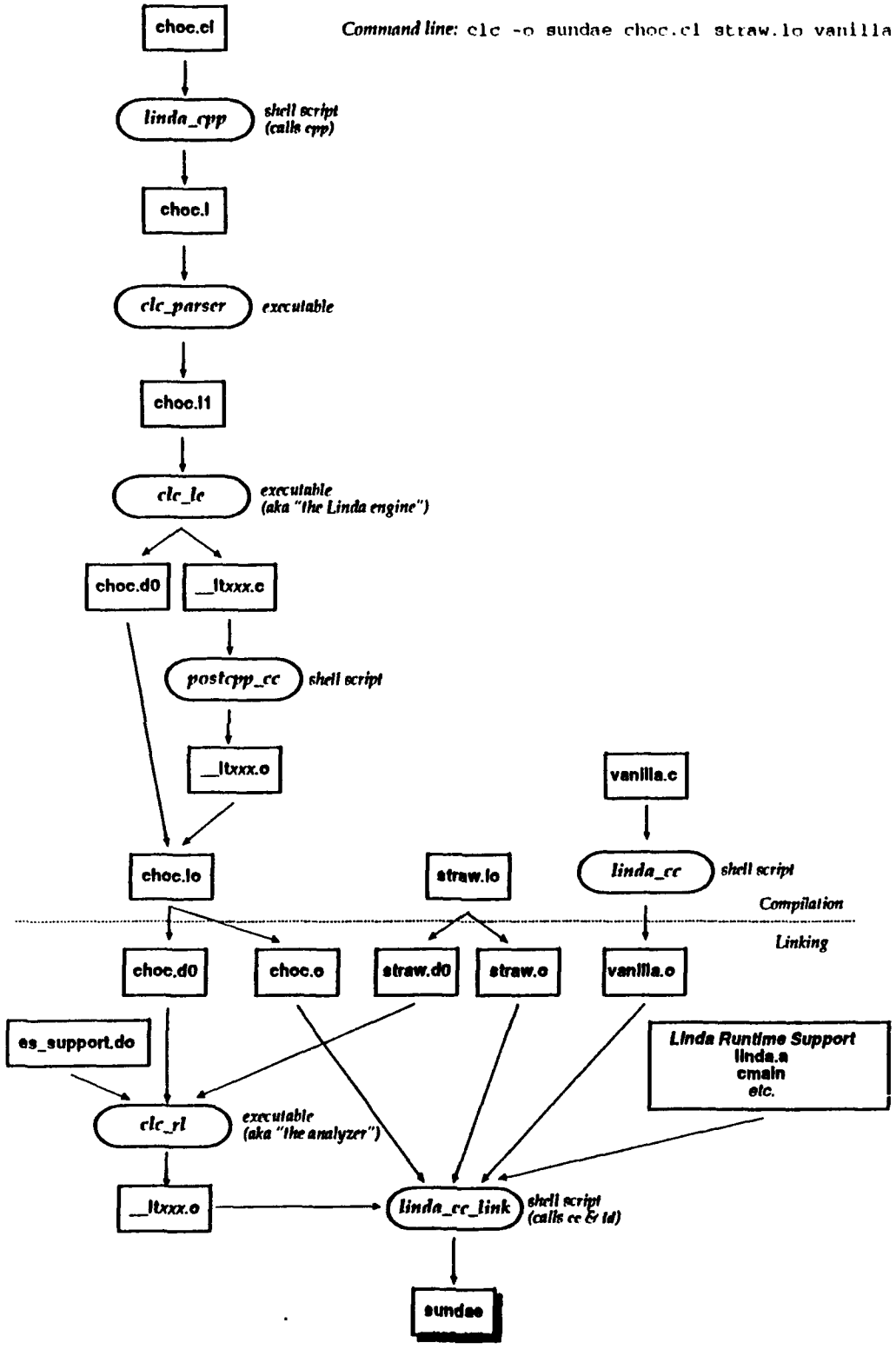

Figure 1: C-Linda Structure

above. Essentially, these functions contain only calls to appropriate routines in Linda's run-time library. Once the intermediate functions have been compiled (again using the native C compiler), the standard system linker is used to produce the final executable. It should be noted that the linking procedure is fully compatible with construction of multi-language programs, including those written in C++ and other languages, so long as all the routines can be linked together under the base-language linking conventions.

The key to Linda's run-time efficiency is its run-time library: both its design and implementation, and the proper choice of run-time library routines for implementations of the tuple space operations in a program. In SCIENTIFIC's systems, the run-time library is implemented as a poly-library that is, as a collection of families of run-time routines which can be used to implement different kinds of tuple space operations. While the Linda associative-matching protocol is very general, analysis of data collected at compile and link times makes it possible to select the most appropriate member of the family applicable to each operation, thereby maximizing run-time efficiency while maintaining exactly the minimal required amount of generality.

In summary, each Linda operation in the user's program is replaced at compile-time by a call to a randomly-named and as-yet non-existent procedure. The pre-linker implements each of these new procedures, by initializing some data structures and invoking the appropriate routine from the Linda run-time library. Since at link-time there is complete knowledge of every tuple space access in the program, it is possible to carry out a substantial amount of analysis and optimization to improve run-time performance.

To begin with, all tuple operations can be divided into groups, based on the sizes, type signatures, and, where known, constant values in the tuples they manipulate. At run-time, only the operations within a given class can interact, thus reducing the costs of associative matching considerably. For example, an in operation involving an 5-field template with a given type signature can never cause a match with a 4-field tuple or a 5-field tuple with a different type signature. In addition, the presence of constants often helps to distinguish among different classes of tuples.

Once tuple operations (and tuple space) are divided into classes, the pre-linker determines the best way to store and manipulate each class at run-time. Consistently-used constant fields (always appearing in both tuples and templates in a class) are not retained at run-time; in effect, tuples are compressed at run-time by pre-matching constants at link-time. For each tuple class, a particular implementation data structure is selected, according to the amount and nature of matching that will be required at run-time. The simplest classes can be implemented using counters or stacks, depending on whether or not any data copying is required. Other classes are implemented using data structures such as queues, private hash tables, or distributed hash tables.

5.2 C++-Linda Compilation

C++-Linda compilation is more complicated due to the presence of C++ objects, which are properly understood and implemented only in a context of C++. For example, to

copy C++ objects in the C environment, we have to know the internal structure (memory layout) of objects. As this structure varies from compiler to compiler (C++ standards do not specify a layout format for objects, and there is no industry standard), and architecture to architecture (due to alignment requirements, among other things), it is not possible to write generic libraries for this purpose. Our solution to this problem will be to generate object manipulation operations in C++ at compile time, and then have the run-time library (written in C for portability) call these automatically-written routines for services on objects. This way the basic steps in compilation remain the same.

Copying an object into the tuple space is different from copying other aggregates (arrays, Fortran common blocks). We cannot, as we stated before, simply copy a chunk of bytes. Instead, it involves two steps: instantiation of an object of same class in the tuple space and copying the required data fields. In C++, only member functions and the functions declared as "friends" to the class can access private data of a class. So, if we do the data transfer within the context of C++, the transfer function must be a member or a friend of the class.

Interaction of objects of two different classes that have an inheritance relationship is another issue. In C++, if the class B inherits from the class A, then it is valid to copy B to A but not A to B. These asymmetric relationships are new to Linda, and has to be dealt with using additional care.

In summary, while our implementation of C++-Linda will be able to share a considerable amount of technology with existing C-Linda implementations, it will still be necessary to develop some innovative additions to support the essential features of object orientation.

6 Conclusions

We have done a comprehensive study of current research on object-oriented parallel languages. The base language of the object-oriented programming language is decided by many factors involving technical, implementation, and commercial issues. We found that object-oriented properties can be effectively supported within a single logical tuple space. We have designed a C++ based Linda language that combines the advantages of both object-oriented technology and portable parallelism. We found that Linda tuple space operations combines naturally with object orientation. There are some intricacies of C++-Linda that are still unresolved, but they are minor in nature and can be effectively resolved only after a prototype development. We have found a freely available parser to use in our prototype.

References

[1] A.S.Grimshaw. Easy-to-use object oriented parallel programming with mentat. Technical Report CS-92-32, Department of Computer Science, University of Virginia, Charlottesville, 1992.

[2] Rohith Chandra, Anoop Guptha, and John L. Hennessy. Cool: An object-based language for parallel processing. *Computer*, August 1994.